REMUS: A Security-Enhanced Operating System

MASSIMO BERNASCHI
Istituto Applicazioni del Calcolo, CNR
and
EMANUELE GABRIELLI and LUIGI V. MANCINI
Università di Roma

We present a detailed analysis of the UNIX system calls and classify them according to their level of threat with respect to system penetration. Based on these results, an effective mechanism is proposed to control the invocation of critical, from the security viewpoint, system calls. The integration into existing UNIX operating systems is carried out by instrumenting the code of the system calls in such a way that the execution is granted only in the case where the invoking process and the value of the arguments comply with the rules held in an access control database. This method does not require changes in the kernel data structures and algorithms. All kernel modifications are transparent to the application processes that continue to work correctly with no need of source code changes or recompilation. A working prototype has been implemented as a loadable kernel module for the Linux operating system. The prototype is able to detect and block any attacks by which an intruder tries to gain direct access to the system as a privileged user.

Categories and Subject Descriptors: D.4.0 [**Operating Systems**]: General; D.4.6 [**Operating Systems**]: Security and Protection; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms: Design, Security

Additional Key Words and Phrases: Access control, Linux, privileged tasks, system calls interception, system penetration

1. INTRODUCTION

Most of the techniques currently in use for intrusion detection are based on some form of analysis of audit data and system log files [Intrusion Detection Systems 1999; Lunt 1993]. The idea is to examine these data sources for evidence of operations, or techniques known to be used in particular types of attacks. Intrusion detection methods that look for these kinds of indicators are known as signature-based methods. A second approach, known as profile-based methods,

Authors' addresses: M. Bernaschi, Istituto Applicazioni del Calcolo, CNR, Viale del Policlinico 137, 00161 Rome, Italy; email: massimo@iac.rm.cnr.it; E. Gabrielli, L. V. Mancini, Dipartimento di Scienze dell'Informazione, Università di Roma "La Sapienza," 00198 Rome, Italy; email: lv.mancini@dsi.uniromal.it.

tries to spot anomalies (e.g., off-hours root login) and periodically checks the system configuration looking for unexpected changes (e.g., unknown users having special privileges). For instance, a common practice is to perform on a regular basis a comparison between the properties (size, access permissions, etc.) of system files and a reference list. This procedure is aimed at spotting *Trojan horses*, that is, programs left by an intruder which look harmless but allow him or her to take full control of the system. The main advantage of these methods is their being *nonintrusive*; that is, they do not require changes to the operating system (OS) or system commands. However, very often they detect the attack when it is already over. Indeed, research in this area has focused mainly on detecting intrusions after the attacks successfully complete, rather than preventing them from happening. Intrusion prevention could be achieved if the OS could monitor every system call made by every process, and prevent malicious or unexpected invocations of system calls from being completed. For example, the OS could build a profile of the legal invocations of system calls (i.e., security rules) for each application process so that the OS itself could immediately detect a security violation by monitoring the system calls invoked during the provision of the service, and thus preventing intruders from breaking system security.

An analysis of the system calls is crucial to design and implement the system call monitoring in an effective way. We present a complete classification of the UNIX system calls according to their level of threat. The system call analysis described considers various threats including penetrations with full control of the system, and denial of service attacks. However, the focus of our considerations, and the design of the prototype presented, refer mainly to threats by which an intruder tries to gain direct access to the system as a privileged user (e.g., root). Our results identify and isolate a subset of system calls and a subset of tasks critical for system security and hence allow reduction of the monitoring overhead and the development and maintenance effort.

As an application of this methodology, we have developed the REMUS (REference Monitor for UNIX Systems) prototype for monitoring those system calls that may be used to subvert the execution of privileged applications. REMUS employs a simple mechanism for system call interception at the OS kernel level and requires minimal additions to the kernel code and no change to the syntax and semantics of existing system calls. Basically, the system call execution is allowed only in the case where the invoking process and the value of the arguments comply with the rules kept in an access control database (ACD) within the kernel. Common penetration techniques that involve tricking the system into running the intruder's own code in privileged mode are blocked by this approach. In particular, REMUS blocks buffer overflow attacks before they can complete. Note that these are just examples of possible attacks, since our approach intends to protect against any technique that allows an attacker to hijack the control of a privileged process. In addition, our solution allows several response options to be taken into consideration to handle the attack.

The key issues addressed by this work can be summarized as follows.

1. Provide a complete analysis of the critical system calls from the security viewpoint;

2. detect illegal invocation of critical system calls before they complete so as to prevent attackers from hijacking control of any privileged process;
3. allow an efficient check of the argument values of the system calls;
4. implement a secure OS by means of lightweight extensions of the kernel, in particular without requiring changes in existing data structures and algorithms; and
5. support, thanks to the immediate detection of possible attacks, other extensions of the OS to confine and tolerate intrusive processes running together with legitimate processes. This could allow a safe analysis of the attack while the intrusion is in progress.

This article is a revised and extended version of Bernaschi et al. [2000]. In particular the major extensions include:

1. a detailed analysis of the system calls used to create, load, and delete loadable kernel modules;
2. the design and kernel implementation of a scheme to prevent a subverted privileged application from loading a malicious kernel module. The extended system maintains a digital signature of the executable code of the legal modules which are the only ones that can be loaded and executed;
3. a completely new reference monitor implemented as a loadable kernel module. In contrast, in Bernaschi et al. [2000] only a kernel patch implementation which required a simpler analysis and implementation was described; and
4. the access to the ACD by means of the UNIX standard virtual file system `/proc`. The system administrator sees the ACD as a directory; each file contains the set of rules for a given critical system call. This new feature allows the system administrator to access the ACD through the UNIX standard file system interface.

The rest of the article is organized as follows. Section 2 describes the result of the analysis which is the foundation of the present work. Section 3 reviews the main ideas behind our proposal. Section 4 gives details about the algorithms and data structures used in the current implementation in the Linux kernel. Section 5 reviews related approaches presented in the literature. Section 6 concludes the paper and discusses future activities.

2. PROBLEM ANALYSIS

Any modern operating system makes a distinction between ordinary and privileged tasks. One of the duties of the OS is to prevent ordinary tasks from compromising system security. So the focus of our attention is on the privileged tasks that are dangerous if subverted since, by default, the OS trusts them.

The present work is empirical in nature. We have analyzed the purpose and the implementation of the complete set of Linux system calls, we have studied how each of them accesses the security-sensitive data structures, and we have classified the system calls that might be invoked by a subverted privileged task. We have generalized our observations to be able to present arguments and

examples to substantiate the following claim: by adding access control tests to a small number of system calls and by monitoring a subset of the processes, the protection against subverted privileged tasks is complete and cannot be bypassed by executing unprotected system calls. This result reduces the cost of monitoring the system calls since the invocation of most of them is not checked. The analysis described in this section refers to the Linux OS, however, most of the results apply to other flavors of UNIX as well.

A typical class of attacks that try to subvert privileged tasks are those based on buffer overflow, briefly described below.

2.1 The Buffer Overflow Attack

The lack of array bounds checking in C makes it possible to overflow a memory buffer beyond its boundaries. By means of widely known techniques [Aleph One 1996; Conover and the w00w00 Security Team 1999; Mudge 1996], a malicious user may inject executable code in memory buffers belonging to the stack or the data segment of a process P. In addition, if the attacker manages to change the return address of a function, the fragment of code he injected is executed by P. Obviously the process P corresponding to the buggy program maintains its "status" including special privileges (if any). As a consequence, if the technique is successfully applied to a privileged process and the fake code is used, for instance, to start the execution of an interactive shell, the attacker gains access to a privileged shell. System software components, that is, commands, daemons, and libraries written in C, rarely perform all the necessary checks before invoking functions like `sprintf` or `strcpy` that may result in a buffer overflow. This feature makes them an obvious target of buffer overflow-based attacks especially if these components generate processes running with superuser privileges.

2.2 Privileged Tasks

In Linux a privileged task may belong to one of the following three categories.

Interactive. This is any process started "by hand" by the system superuser. Both the user identifier (UID) and the effective user identifier (EUID) are equal to 0. No additional threat is generated by such a task since the user who starts it already has full control of the system;

Setuid. This is a task employing the UNIX mechanism of the setuid bit to grant ordinary users special privileges on a temporary basis. For this kind of task the pair of user identifiers (UID, EUID) has values UID > 0 and EUID $= 0$. So a task can be identified in the Linux kernel as setuid to root by means of the simple macro

```
#define IS_SETUID_TO_ROOT(proc) !((proc)->euid)&&(proc)->uid.
```

For specific OSs, other information may be more suitable to identify a setuid process. For example, in the AIX OS a good candidate appears to be the LOGIN UID which is assigned during the initial user authentication and never modified. Our choice is motivated by the consideration that the pair (UID,

EUID) is present in all UNIX flavors. This should ease the extension of our work to other UNIX platforms.

Background. This is, usually, either a daemon process started at boot time or a task started periodically by the `cron` daemon on the root behalf. Both the UID and the EUID are equal to 0. The difference from interactive tasks is described in Subsection 2.2.1.

2.2.1 *Background Root Tasks.* On a typical UNIX system there are always root programs running in "background." Most of the time, the system administrator neither starts them directly (i.e., during an interactive session) nor controls their execution, therefore such "unattended" programs are preferred targets of attacks.

According to Stevens [1998], it is possible to group them as follows.

1. Daemon processes originated directly by the system initialization scripts. Network servers such as the `inetd` super-server, and Web server, and mail server (e.g., `sendmail`) are often started in this way. Another example is the `syslogd` daemon.
2. Network servers started by the `inetd` superserver to fulfill requests for services such as remote access (telnet), file transfer (FTP), and so on.
3. Programs executed on a regular basis by the `cron` daemon. The `cron` daemon itself is started at boot time (i.e., it belongs to Group 1).
4. Programs executed just once in the future by means of the `at` command. Actually these programs can be considered as a special case of the previous group.
5. Programs started in background during an interactive session (i.e., with an & at the end of the command line). This is done mostly for testing purposes or for restarting a daemon that was terminated for some reason.

These programs do not have a controlling terminal. To tell them apart from root programs running in interactive mode, we resort to the macro

```
#define IS_A_ROOT_DAEMON(proc) !((proc)->euid)&&((proc)->tty==NULL).
```

Here, the first logical clause checks whether the process runs with root privileges (EUID = 0) whereas the second checks whether the process has a controlling terminal. Following Stevens [1998] and Comer and Stevens [1998] we assume that a daemon never needs a control terminal. Note that:

—there is no system primitive that allows a task to reacquire a control terminal. As a consequence a subverted daemon cannot bypass our control mechanism trying to behave as an interactive task; and

—a daemon can still open a terminal device (e.g., `/dev/tty` or `/dev/console`) to log error messages.

2.3 System Call Analysis

The analysis presented in this section identifies the system calls that may jeopardize system security. The system calls available in Linux 2.2 have been

Table I. System Call Categories

Group	Functionality
I	File system, devices
II	Process management
III	Module management
IV	Memory management
V	Time and timers
VI	Communication
VII	System info
VIII	Reserved
IX	Not implemented

Table II. Threat Level Classification

Threat Level	Description
1	Allows full control of the system
2	Used for a denial of service attack
3	Used for subverting the invoking process
4	Harmless

grouped in categories according to their functionality as reported in Table I. In addition, each system call has been classified according to Table II which portrays the essential features of the four levels of threat considered. By convention, less dangerous system calls are assigned a larger threat level number. This classification corresponds to a threat hierarchy, since a system call classified at threat level n may be employed also to carry on an attack at threat level m if $m \geq n$. For example, if a system call allows the attacker to gain access to a privileged shell, then the attacker has full control of the system (threat level 1). In this case it is trivial to carry on a denial of service (threat level 2), and so on. Note that the reverse does not hold. For example, threat level 2 classifies the system calls that may be used for a denial of service attack, but that do not give an intruder full control of the system (e.g., the `reboot` system call). Threat level 3 contains the system calls that may be used to subvert the behavior only of the particular process which is invoking that system call, but cannot directly jeopardize the security of the rest of the system (e.g., the `pause` system call can block a particular process running a particular service of the system).

There are a number of possible sublevels within each class defined in Table II. For instance, a threat can be remote or local. Typically an intruder who does not have direct access to a system can attack only daemon tasks, whereas a local attacker may try to hijack setuid programs, too. Another possible difference is between direct and indirect threats. An example of a direct threat is the execution of an interactive shell with root privileges. An indirect threat is the retrieval of the encrypted passwords that may be used for an offline dictionary attack. In this article we address the issues raised by system calls classified as threat level 1 regardless of the origin of the attack. The analysis and the distinction in four classes of threats may be used for future extensions since the system architecture we describe in Section 3 is not limited by the present choice.

Table III. System Calls Classified by Threat

Threat	Group	System Calls
1	I	open, link, unlink, chmod, lchown, rename, fchown, chown, mknod, mount, symlink, fchmod
	II	execve, setgid, setreuid, setregid, setgroups, setfsuid, setfsgid, setresuid, setresgid, setuid
	III	init_module
2	I	creat, umount, mkdir, rmdir, umount2, ioctl, nfsservctl, truncate, ftruncate, quotactl, afs_syscall, dup2, flock
	II	fork, brk, kill, setrlimit, reboot, setpriority, ioperm, iopl, clone, modify_ldt, adjtimex, sched_setparam, vfork, vhangup, sched_setscheduler, vm86, vm86old
	III	delete_module
	IV	swapon, swapoff, mlock, mlockall
	V	stime, settimeofday, nice
	VI	socketcall, ipc
	VII	sethostname, syslog, setdomainname, _sysctl
3	I	read, write, close, chdir, lseek, dup, fcntl, umask, chroot, select, fsync, fchdir, _llseek, _newselect, readv, writev, poll, pread, pwrite, sendfile, putpmsg, utime
	II	exit, waitpid, ptrace, signal, setpgid, setsid, sigaction, ssetmask, sigsuspend, sigpending, uselib, wait4, sigreturn, sigprocmask, personality, capset, rt_sigreturn, rt_sigaction, rt_sigprocmask, rt_sigpending, rt_sigtimedwait, rt_sigqueueinfo, rt_sigsuspend, sched_yield, prctl
	IV	mmap, munmap, mprotect, msync, munlock, munlockall, mremap
	V	pause, setitimer, nanosleep
4	I	oldstat, oldfstat, access, sync, pipe, ustat, oldlstat, readlink, readdir, statfs, fstatfs, stat, getpmsg, lstat, fstat, olduname, bdflush, sysfs, getdents, fdatasync
	II	getpid, getppid, getuid, getgid, geteuid, getegid, acct, getpgrp, sgetmask, getrlimit, getrusage, getgroups, getpriority, sched_getscheduler, sched_getparam, sched_get_priority_max, sched_get_priority_min, sched_rr_get_interval, capget, getpgid, getsid, getcwd, getresgid, getresuid
	III	get_kernel_syms, create_module, query_module
	V	times, time, gettimeofday, getitimer
	VII	sysinfo, uname
	VIII	idle
	IX	break, ftime, mpx, stty, prof, ulimit, gtty, lock, profil

The details of system call classification are summarized in Table III, and are briefly discussed below. No system call in Groups IV to IX can be used to gain control of the system. For instance, system calls in Group IX immediately return the error `ENOSYS`, whereas primitives in Group VIII (reserved) cannot be invoked by a generic user process (even if it is a privileged process). The system calls belonging to Group VII (system info) return or set information such as the host or domain name which although important do not allow an attacker to take control of the system (these become critical if denial of service (DOS) attacks

are considered). Similar considerations apply to class V (i.e., system calls related to the management of time and timers). The two system calls in Group VI are the basis of any interprocess communication (either local or remote). They do not access any security-related resource. System calls in Group IV (memory management) are especially critical for DOS attacks (e.g., `mlock` disables paging for some parts of memory). However, since the resources controlled do not include the file system or the process privileges, these primitives are not critical for system security.

As for the system calls in Group III (module management), a subverted process may use them for loading a malicious module (e.g., a module that is not listed in /lib/modules). Our investigation shows that `init_module` is the only primitive that reaches threat level 1 since no module can be activated without invoking `init_module` first.

The largest set (78) of system calls is related to process management (Group II). Among these primitives 10 reach the highest level of danger for system security. The `execve` can be used to start a root shell. The other 9 primitives set user and group identifiers. It is worth noting that `capset` allows a process only to restrict its capabilities, thus it is considered level 3.

The 12 system calls related to the file system (Group I) classified as threat level 1 require special attention. An intruder having access to a CD-ROM connected to the "victim" system may build a file system that contains a fake `/bin` with Trojan horse programs. Then, by means of the invocation

```
mount(‘‘/dev/hdb’’,‘‘/bin’’,‘‘ext2’’,MS_MSG_VAL,NULL)
```

the intruder may cover the legitimate `/bin` directory with his fake `/bin`.

Although less common, there are exploits by no means less dangerous than those based on the "classic" *shellcode*. It is apparent that

```
chmod(‘‘/etc/passwd’’,0666),
chown(‘‘/etc/passwd’’,intruder,intruder_group)
rename(‘‘/tmp/passwd’’,‘‘/etc/passwd’’)
```

compromise the OS authentication mechanisms. However, it is necessary to consider *chains* of system calls as well. For instance, `chown` and `chmod` primitives can be used in a two-step attack to create a setuid shell:

```
chown(‘‘myshell’’,root,root_group)
chmod(‘‘myshell’’,4755).
```

Whereas the sequence

```
unlink(‘‘/etc/passwd’’)
link(‘‘/tmp/passwd’’,‘‘/etc/passwd’’)
```

produces the same result as the `rename` primitive.

The above chains of system calls can be executed either in a single attack or in two successive and independent attacks. Other forms of dangerous chains of system calls (e.g., a combination of `open` and `write`) allow the creation of a new account with UID = 0 or the creation of a fake *.rhosts* file in the root home directory. However, attacks of this form must be carried on within the context

Table IV. Threat Level 1 System Calls

System Calls	Dangerous Parameter
`chmod, fchmod`	A system file or a directory
`chown, fchown, lchown`	A system file or a directory
`execve`	An executable file
`mount`	On a system directory
`rename, open, mknod`	A system file
`link, symlink, unlink`	A system file
`setuid, setresuid, setfsuid, setreuid`	UID set to zero
`setgroups, setgid, setfsgid, setresgid, setregid`	GID set to zero
`init_module`	Modules not in `/lib/modules`

of a single subverted task since the `write` system call needs a file descriptor returned by `open`. Thus the subverted task invokes at least two system calls: a first one to open the file and a second one to modify it. In this case, in accordance with Goldberg et al. [1996] we assume that it is necessary to monitor only the `open` primitive. This is why the `write` system call is not considered threat level 1.

We recall that the checks are enforced on setuid or daemon programs only in the case where they invoke critical system calls keeping EUID = 0 (i.e., with their special privileges). Since these programs usually access files in write mode with EUID equal to the owner of the file, the access to user files does not need to be authorized.

A detailed study of setuid and daemon programs has been carried out to define which directories and files they need to write for "legal" purposes. This has been realized by both source code inspection and analysis of the results produced by the `strace` command which intercepts and records the system calls invoked by a process.

Table IV summarizes the results of the analysis.

3. SYSTEM ARCHITECTURE

A radical approach in addressing the security issues raised by privileged tasks is to control the execution of any system call. A widely used abstraction to define such control is the concept of the reference monitor described in Ames et al. [1983]. The basic idea is that the only path the processes can take to access the system calls is through the reference monitor shown in Figure 1. The reference monitor consists of two main functions: the reference function and the authorization function. The reference function is used to make decisions about whether to permit or deny a system call request based on information kept in an access control database (ACD). The ACD conceptually contains entries or access control rules in the form of a process, system call, or access mode. The access control rules in the ACD capture conditions on both the system calls and the values of their arguments. For instance, an access control rule of the `execve` system call could specify in the access mode field the list of executable files that the invoking process can execute, that is, the files that the invoking process can legally pass as an argument to `execve`. Such a feature prevents an improperly registered privileged process from starting "dangerous" programs such as an interactive shell. Note that the checks on a system call which access the file

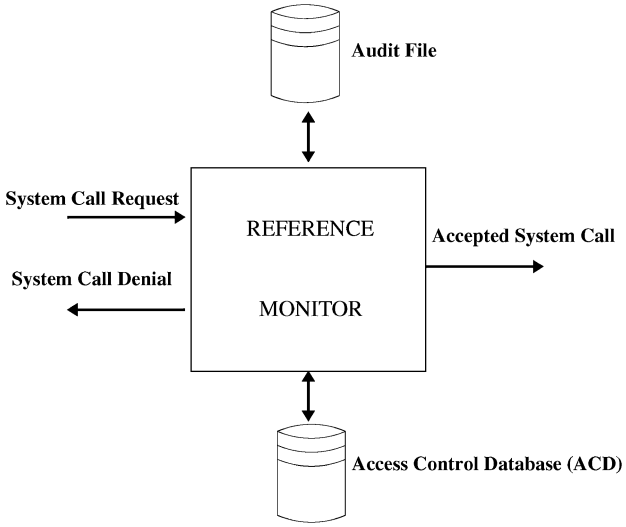

Fig. 1. Reference monitor.

system are not based simply on the file name passed as the parameter. Actually, the checks refer to information stored in the *inode* of the file. As explained in Section 4, tests based on file names can be bypassed by renaming the file parameter. The ACD is not part of the reference monitor, but all modifications to such a database are controlled by the reference monitor by means of its second component, the authorization function. This function is used to monitor changes to individual access control rules.

One of the fundamental principles of a reference monitor is its completeness, meaning that all accesses must be mediated by the monitor. Unfortunately, implementation at the system call entry point level of this principle has a high cost which is constant regardless of the system call [Sekar et al. 1999]. Moreover, a fine-grain control of over 200 primitives is a major development (and maintenance) effort. We have reduced the overhead and simplified the development of a prototype relying on the analysis of Section 2. This allows the identification of a subset of system calls and a subset of tasks (the noninteractive privileged tasks) that need to be monitored to offer complete protection against attacks aimed at gaining immediate and full control of the target system. In other words, the system cannot be attacked by executing unprotected system calls.

Following this approach, we have built a prototype that detects and blocks attacks based on the subversion of a privileged task (e.g., attacks based on buffer overflow) before they can complete. The prototype implements the reference monitor functions and in particular the system calls interception inside the kernel of the Linux OS. By means of checks made by the OS kernel before the system call is completed, it is possible to prevent any possible side effect of system calls that intruders currently exploit for their attacks.

It is worth noting that the checks performed inside the system calls do not require changes in the existing kernel data structures and algorithms. Hence, all the kernel modifications are transparent to the application processes that

```
/* execve_acd */
typedef struct setuid_proc_id {
   char comm[PROGNLEN];  /* look at /usr/src/linux/include/linux/sched.h */
   unsigned long count;
} suidpid_t;
typedef struct  setuid_program {
   suidpid_t suidp_id;
   suidp_t   * next;  /* next program      */
} suidp_t;
typedef struct exe_file_id {
   __kernel_dev_t   device; /* device number     */
   unsigned long    inode;  /* inode number      */
   __kernel_off_t   size;   /* size              */
   __kernel_time_t  modif;  /* modification time */
} efid_t;
typedef  struct executable_file {
   efid_t   efid; /* info for file identification */
   int prog_nr; /* number of programs that can invoke this exe */
   suidp_t *programs; /* list authorized programs */
} efile_t;
typedef struct executable_file_list  {
   efile_t  lst[NR_EXE];
   unsigned int total; /* total number of exe in the list */
} eflst_t;
```

Fig. 2. The layout of execve_acd access control data structures.

continue to work correctly requiring neither changes of the source code nor recompilation.

4. IMPLEMENTATION

This section describes the enhancements of the Linux kernel that block any attempt at hijacking control of a privileged process. In particular, this section addresses the attacks by which an intruder tries to gain immediate access to the system as a privileged user (i.e., root). According to this choice, the prototype monitors those system calls invoked by root processes that may compromise the security and integrity of the operating system. Of course in a system that maintains user-sensitive information, a subverted program should never read or modify user information too.

The description includes the authorization functions, the data structures added to the OS kernel to implement the access control database, the new system call to read, write, and update the ACD, and the reference functions.

4.1 The Authorization Functions

The ACD contains a section for each system call under the control of the Reference Monitor. For instance, the REMUS prototype maintains a `setuid_acd` data structure to check the access to the `setuid` system call and an `execve_acd` data structure to check the access to the `execve` system call. The layout of the `execve_acd` data structure is shown in Figure 2. The `execve_acd` is composed of two arrays of `eflst_t` structures.

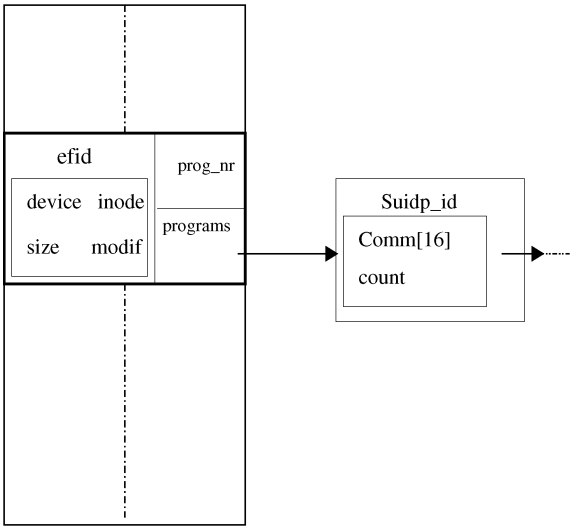

Fig. 3. The layout of the *admitted* data structure.

`admitted`: An executable file F has an entry in this structure if at least one privileged program needs to execute F via an `execve`. The information stored in the entry is the list of all privileged programs that may invoke F.

`failure`: This list keeps a log of the unauthorized attempts (i.e., not explicitly allowed by the `admitted` data structure) of invoking `execve` by any privileged process.

Figure 3 shows the `admitted` data structure which is an array where each element refers to an executable file and points to a list of privileged programs that can execute that file. The `failure` data structure (not shown in the figure) is similar to the `admitted` data structure, but grows dynamically as long as the kernel intercepts unauthorized invocations by privileged programs. Currently it is kept just for auditing purposes. Each element of the `admitted` data structure contains three fields: `efid`, `proc_nr`, and `programs`.

`efid`: This field identifies the executable file F. The information stored in `efid` is:

`device`: the device number of the file system to which file F belongs;

`inode`: the inode number of file F;

`size`: the length in bytes of file F; and

`modif`: the last modification time of file F.

The pair `device` and `inode` identify file F in a unique way within the system whereas `size` and `modif` allow the detection of unauthorized modifications of the file contents.

`prog_nr`: This field indicates the length of the programs list, that is, the number of privileged programs that can invoke file F.

programs: This is a pointer to the list of privileged programs that can execute file F. Each element, named suidp_id, of the list contains two fields: comm and count. The string of characters comm keeps a copy of the name of the privileged program as it appears in the comm field of the process table. The field count is used for statistics and indicates the number of invocations of file F by process comm.

We have introduced a new system call, sys_aclm, which allows the interactions with the OS kernel for reading and modifying the information kept in the ACD. The sys_aclm can be invoked only by interactive root processes with EUID = 0 and UID = 0. These restrictions are required to prevent a subverted setuid or root daemon from tampering with the ACD. Since distinct root processes can access the ACD, conflicts can arise. Hence our new primitive enforces a concurrency control mechanism among the conflicting processes. In particular, a lock variable called write_pid is used to implement mutual exclusion. To avoid race conditions on write_pid itself, this variable must be checked and updated atomically. This has been achieved by means of a kernel function called atomic_access which resorts to the Intel Architecture instruction for atomic exchange: *xchg*.

Actually, the sys_aclm system call is a common frontend for five different operations which are briefly described below.

PUT(entry, list) adds entry in the specified list of the ACD.

GET(entry, list) reads entry from the specified list of the ACD.

DELETE(entry, list) removes entry from the specified list of the ACD. If entry is NULL, then it removes all entries of the list.

PUTHEADERACL(header-acl) stores the header of the ACD in kernel memory space. It is useful when the system is started the first time after installation.

GETHEADERACL(local-header-acl) retrieves the header of the access control database from the kernel space and stores it in the local variable local-header-acl.

The system administrator can manage the ACD resorting to the sys_aclm system call through a new command, named aclmng. For instance, the following command allows the program sendmail to write any file in the directory /var/spool/mail:

```
#aclmng -v --open -D /var/spool/mail /usr/sbin/sendmail,
```

whereas to prevent a subverted task from loading a malicious parport module, the legal module may be signed by means of the MD5 algorithm [Rivest 1992] with the command (see Section 4.2 for the details):

```
#aclmng -v --init_module /lib/modules/2.2.12-9/parport.o.
```

The contents of the ACD can be listed by means of the /proc file system. There is a node for each system call and a header which gives the current configuration of the control mechanism (e.g., system calls under

```
#cat /proc/acd/acd
 REMUS Version 1.0 : ACD Header
 writerpid: 0
 setuid encrypted root passwd:
 syscall execve            11      ctlmask=3   check,block,nodebug
 syscall open              5       ctlmask=17  check,block,debug
 syscall setuid            23      ctlmask=3   check,block,nodebug
 syscall initmodule        127     ctlmask=0   nocheck,noblock,nodebug
 syscall link              9       ctlmask=17  check,block,debug
 ..............................................................
```

Fig. 4. The header of the ACD as shown by the proc filesystem.

control, blocking, or debugging flags). An example of the output is shown in Figure 4.

The addition of a node to the /proc file system requires filling a structure proc_dir_entry with suitable values and calling the proc_register primitive to hook the new node to its parent in the /proc hierarchy. For instance, to add the open node:

```
static struct proc_dir_entry proc_acl_open = {
   PROC_ACL_OPEN,      /* i-node number; it is defined in an enum */
   4,                                          /* node name length */
   "open",                                           /* node name */
   S_IFREG | S_IRUGO,         /* file type and access permissions */
   1,                                            /* number of link */
   0, 0,                                    /* UID, GID of the owner */
   0,                      /*  The size of the file reported by ls */
   &acl_openpiops /* pointer to the struct with the specific ops. */
};

if(proc_register(&proc_acl, &proc_acl_open)<0) {
   /* Error ... */
}
```

In the present prototype, the entries of the ACD cannot be modified via the /proc interface. One of the possible enhancements we are considering is to allow the configuration of the ACD and the activation of the control mechanism by means of the standard sysctl command. This requires adding to /proc/**sys** an acd node and subnodes to /proc/**sys**/**acd** for all system calls under control. In this way we could get rid of the aclmng command.

4.2 The Reference Functions

This section presents some examples of the extensions introduced to control the system calls defined in Section 2.3 as threat level 1.

execve. As shown in Figure 5, the reference function is invoked at the beginning of this system call after the file has been opened. The check_rootproc() function (that we report in the appendix) authenticates the root process that invokes execve and checks in the access control database whether the calling

```
/*
 * sys_execve() executes a new program.
 */
int do_execve (char * filename, char ** argv, char ** envp, struct pt_regs * regs){

    ...
    dentry = open_namei(filename, 0, 0);
    retval = PTR_ERR(dentry);
    if (IS_ERR(dentry))
      return retval;
    ...
    retval = prepare_binprm(&bprm);

    /********** REMUS PATCH ************************************/
    rc=check_rootproc(bprm.dentry->d_inode);
    if ((rc==EXENA) || (rc==EFNA)) {
      printk(BOP_LEVEL"REMUS kernel:do_execve psuid %s no
             authorized to exec file %s\n",current->comm,filename);
      printk(BOP_LEVEL" by euid %d uid %d\n",
             current->euid,current->uid);
      if (rc==EXENA)
        printk(BOP_LEVEL" EXE NO AUTHORIZED\n");
      else  printk(BOP_LEVEL" EXE NO AUTHENTICATED\n");
      return rc;
    }
    /***********************************************************/
    ...
    if (retval >= 0)
      retval = search_binary_handler(&bprm,regs);
    if (retval >= 0)
       /* execve success */
        return retval;
    ...
}
```

Fig. 5. The "patch" to the execve system call.

process has the right to execute the program whose name is passed as the first parameter. The system call execution is denied when check_rootproc returns one of the two values:

> EXENA: The calling process is not authorized to execute the requested program. That is, the program name is not present in the access control database or the calling program is not listed in the programs field of the admitted list in the access control database.
>
> EFNA: The calling process is authorized to execute the requested program, but the file is not authenticated; for instance, the modification time or the size does not match.

In the check_rootproc function, if the calling process does not run with root privileges (EUID = 0), then no further check is performed and the execve proceeds normally. Otherwise, the service is provided if and only if the permission is explicitly contained in the access control database.

Figure 5 refers to the version of REMUS in which the interception is carried on within the code of the system call. Actually, there is another version of REMUS that overwrites the system call table to redirect the execution to routines we call REMUS wrappers (see Ghormley et al. [1998] for details of this technique). Each wrapper includes the security checks required for the corresponding system call. If the outcome of the checks is positive, the real system call is invoked, otherwise the execution is denied giving the same errors as the patch version.

Although the patch version requires a (limited) modification of the original kernel code, there are a number of advantages with respect to this second interception mechanism.

—The interception overhead may be reduced to the bare minimum since there is no additional abstraction layer.
—Most information required by the control mechanism is available and can be used directly so there is no duplication of work. For instance, the inode number of the file that the system should *exec* is determined by the function `open_namei` only once.
—There is no way to bypass the control mechanism since it is part of the system call execution path.

The main drawback of the patch version is the lack of portability and modularity. The security checks are spread in the system calls' code. As a consequence, porting REMUS to new versions of the Linux kernel requires more attention.

`setuid`. For the `setuid` system call, the authentication of the root processes is the same as in the `execve` case. A user running a setuid program which attempts to invoke `setuid(0)` to assign UID = 0 to itself, is forced to type the root password. The keyed password is then compared to an encrypted copy kept in the access control database. In the case of a password mismatch the `setuid(0)` invocation is denied. An example of a program that is controlled with this mechanism is `su`, a setuid program which runs a shell with a substitute user (and group) ID. The other system calls that manage user and group identifiers are controlled in the same way.

`chmod`. An additional check is performed on the `filename` argument if `chmod` is invoked by a background or setuid process. If `filename` refers to a regular file or a directory, the operation is denied. This means that the operation is allowed if `filename` refers to a device registered in the ACD. Important programs, such as the *X* server (which is a background root process) would not work without this distinction. These considerations apply to `fchmod`, `chown`, `fchown`, and `lchown` as well.

`open`. Several setuid programs or root daemons, for good reasons, may open critical files. The present prototype monitors only the access to files in write-mode. We understand that it may be possible to steal sensitive information such as encrypted passwords or the names of "trusted" hosts, but attacks of this type are beyond the scope of the present work. At times it may be necessary to let root programs write any file in specific directories defined in the ACD. This is

the case of directories like /var/mail or /var/spool/mqueue used by sendmail. For directories like /etc a fine-grain control is required. So, a file *foo* belonging to /etc may be modified by a program *prog* only in the case where *prog* and the identifiers (inode and device number) of *foo* are explicitly inserted in the ACD.

init_module. The loading of kernel modules in Linux is a two-step procedure: the create_module system call reserves memory for the module *mymodule* and init_module loads it in memory and calls the initialization entry point of *mymodule*. It is necessary to prevent a subverted task from loading a malicious module (e.g., a module that changes the permission of the /etc/passwd file). One could assume that modules found in "well-known" directories, like /lib/modules, are legal. Unfortunately there is no way for init_module to know the module's origin. To address this issue a signature of the contents of each legal module is stored in the ACD. In init_module the signature of *mymodule* passed as a parameter is computed. If there is no signature for *mymodule* in the ACD or if there is a mismatch between the two signatures, the init_module fails and *mymodule* is not loaded in the kernel.

A widely used technique to define the signature of a data block is to compute its checksum by means of the MD5 algorithm [Rivest 1992]. This approach cannot be applied in a simple way to the object file that contains the module because some parts of it change due to the address's relocation (a step of the loading procedure required to support the dynamic linking of a module or an executable file with shared libraries). A possible solution is to skip the relocatable entries when the MD5 algorithm is applied to the text section of the module object file. The position of the entries that must be relocated is part of the information included in the header of each section of an object file compliant with the executable and linkable format (ELF, the standard object format in Linux). Since the ELF headers are not available within the init_module, it is necessary to maintain the position of the relocatable entries in the ACD. Luckily the number of such entries is such that the size of the ACD does not increase too much.

In addition to the system calls defined in Section 2.3 as threat level 1, two system calls are controlled by REMUS: sys_aclm and delete_module. An attacker could use sys_aclm to remove an access control rule from the ACD, and delete_module to remove the kernel module that implements REMUS itself. For these reasons there are entries in the ACD for these two primitives such that sys_aclm and delete_module (for the REMUS module) cannot be invoked by any root daemon or setuid task.

4.3 Prototype Code Structure

The software prototype is available from ftp://ftp.iac.rm.cnr.it/pub/REMUS and is composed of the parts:

1. a "loadable" module containing the ACD structures, the sys_aclm system call, and the /proc interface;
2. a small set of kernel *patches*. They implement the interception mechanism and consist of new fragments of code added to the existing system calls (there are neither changes nor deletions). Although the patches have been

developed and tested with a specific version (2.2.12–20) of the Linux kernel, we do not expect major problems in porting them to other (newer) versions of the kernel; and

3. the new command `aclmng` (described in Section 4.1).

The system administrator's duties are limited to filling the ACD running the `aclmng` command. The recompilation of existing applications or other software components is not required. Note that filling the ACD on large installations of homogeneous systems is very simple since most of the rules contained in the ACD can be reused with no change. In the case where execution of a system call is denied by the control mechanism a message is sent to the kernel ring buffer. In such a way, the administrator may check the presence of potential attacks by looking at the messages starting with the prefix "REMUS."

4.4 Performance

We expect a very limited degradation of the global performance for a system running our enhanced kernel. There are a number of reasons for this prediction.

— When a process runs in *user* mode, there is no difference at all from a standard system since all new checks are executed in *kernel* mode.

— When a process runs in *kernel* mode, only a limited subset of the processes execute all the checks. A generic user process that invokes a threat level 1 system call undergoes only the following controls within the instrumented system calls:

```
if(IS_SETUID_TO_ROOT(current) || IS_A_ROOT_DAEMON(current) ) {...}.
```

So if the process is neither a root daemon nor setuid to root, it does not execute more than a couple of logical tests (note that the two conditions are evaluated separately simply for the sake of code readability).

— Very few system calls include the additional checks (approximately 10% of the total number of system calls).

— With the exception of the `open` primitive, it is unlikely that a setuid or daemon process invokes any of the instrumented system calls more than once during its lifetime.

— The checks do not require any access to "out of core" data. All the information is resident in the kernel memory.

To assess these considerations, a set of experiments has been executed. We have selected four applications and run them on the same hardware system (a 330 MHz Pentium II with 128 Mb of RAM) with a standard Linux kernel version 2.2.12, then with the patch-based REMUS kernel, and finally with the module-based REMUS kernel. The following four applications were measured: `sendmail, lpr, rsync`, and `X` server. The result of the experiments, reported in detail in Bernaschi et al. [2000], reveals that the difference among the average execution times of the applications is comparable with the standard deviation of the multiple runs. This suggests that the actual impact of REMUS on the global system performance is not noticeable for all practical purposes.

Table V. Execution Time Without `dirent` Cache of the `open` System Call (in Microseconds)

Path	Standard Linux	Patch-Based REMUS	Module-Based REMUS
12	127.10	132.04	238.76
11	119.80	123.28	226.80
10	115.46	117.51	217.99
9	103.57	105.64	197.80
8	94.44	96.70	178.12
7	83.92	85.88	157.76
6	73.38	74.19	136.22
5	64.20	65.70	118.89
4	52.76	54.82	98.08

We have also devised further experiments, in order to evaluate even the minimal overhead due to the reference function performed by REMUS. The experiments focus on a single invocation of a critical system call and measure the interception overhead and the evaluation of the relevant access control rule in the ACD.

Note that, in the module-based REMUS prototype, the system call that requires the highest monitoring overhead is `open`. Indeed, the invocation of `open` requires a double pathname resolution, that is, two invocations of the `namei` routine. First, `namei` is invoked in the security module to find the inode number and the other file information needed for evaluating the access control rules in the ACD. Then, if the comparison succeeds, a second path name resolution is executed in the standard native code of the unmodified system call. Furthermore, in the patch-based REMUS prototype, all critical system calls have a comparable monitoring overhead. In particular, the invocation of `open` requires only a single standard pathname resolution. Indeed, the file information needed for the check is determined during the execution of the standard code of the system call. The execution is intercepted before the actual file is opened, and the available file information is used directly to check against the ACD. Hence, in the patch-based solution the monitoring overhead of the critical system calls is limited.

For supporting these statements, we have carried out some microbenchmarks for the `open` system calls in the three systems: standard Linux, patch-based REMUS, and module-based REMUS. Note that the Linux kernel maintains a cache for the translation of a pathname into an inode. This cache is accessed by `namei`, and is called the `dirent` cache or directory entry cache. As a first experiment, in order to obtain a more precise measurement of the execution time of `open`, the directory entry cache has been disabled by commenting the cache lookup code in the kernel.

Table V reports the average execution time for `open` that we were able to measure when the directory entry cache was disabled. A slower hardware system (a 150 MHz Intel Pentium with 32 Mb of RAM) has been used to increase the absolute value of the measures taken in microseconds by reading the real-time clock register of the microprocessor before and after execution of the system call. The first column reports the length of the path passed as a parameter of `open`. Note that the execution time decreases linearly with the path-length in the three systems.

Table VI. Execution Time of `namei` and Estimation of `open` (in Microseconds)

Path	`Namei`		Module-Based REMUS
	Cache	No Cache	`open`
12	27.60	115.83	151.53
11	24.58	109.41	141.97
10	22.37	105.45	134.91
9	20.15	93.66	124.29
8	18.27	84.92	111.47
7	16.14	74.52	99.38
6	14.40	63.91	86.71
5	12.37	55.08	76.18
4	10.60	43.98	64.70

The measurements related to the module-based REMUS reported in Table V, represent a loose upper bound of the actual execution time. Indeed, the second invocation of `namei` in the module-based implementation, always has a lower computational cost, since the directory entry cache always contains the translation of the pathname into the relevant inode which is loaded after the first invocation. In Table V, the caching function was disabled and both the consecutive invocations of `namei` paid the maximal cost. This situation will hardly occur in practice.

In order to give a better evaluation of the impact of caching on the execution time of `open`, we have measured the execution times of `namei` in the standard Linux as a function of the path-length. The second and third columns of Table VI report the measurements with the directory entry cache enabled, and with this caching function disabled, respectively. Based on these measures, the execution time of `open` in module-based REMUS can be closely approximated by the formula

$$T_{open} \leq moduleBasedOpen_{nocache} - namei_{nocache} + namei_{cache},$$

where $moduleBasedOpen_{nocache}$ indicates the cost of `open` reported in Table V for module-based REMUS. This formula considers the second pathname resolution always executed in cache. The fourth column of Table VI reports the values of the above formula for different path-lengths. Note that the effect of caching mitigates the cost of performing the pathname resolution twice.

If we define the *monitoring overhead* as the ratio of the execution time of `open` in REMUS with respect to the execution time in the standard Linux kernel, then we can conclude that the monitoring overhead per critical system call of patch-based REMUS is about 3%, whereas that of module-based REMUS is about 18%. Note that the extra cost of the pathname resolution ($namei_{cache}$) in the module-based REMUS amounts to less than 18% in the overall cost of `open`.

5. RELATED WORK

Other groups in the recent past have proposed addressing security issues by means of special controls on the values of system call arguments. In Goldberg et al. [1996] a user-level tracing mechanism to restrict the execution environment of untrusted helper applications is described. Our solution is based on a

similar analysis of the potential problems associated with a subset of system primitives, but we control a different set of programs (i.e., root daemons and setuid programs instead of helper applications). We add our additional checks to the system call code mainly for performance reasons but the impact on the existing kernel code is reduced to the bare minimum.

The domain and type enforcement (DTE) is an access control technology which associates a domain with each running process and a type with each object (e.g, file, network packet). At run-time a kernel-level DTE subsystem compares a process's domain with the type of any file or the domain of any process it attempts to access. The DTE subsystem denies the attempt if the requesting process's domain does not include a right to the requested access mode for that type. DTE is a very general approach to mandatory access control; however, it may require deep kernel modifications (about 17,000 lines of kernel resident code and 20 new system calls for DTE-aware applications) [Badger 1995]. Several prototypes of the DTE architecture have been presented in the past. For instance, DTMach and DTOS [Fine and Minear 1993; Minear 1995] are both based on the Mach microkernel and Flask is based on the Fluke operating system developed by the University of Utah. A group sponsored by the National Security Agency (NSA) is currently integrating the Flask architecture into Linux. They have announced the availability of a prototype called SELinux (security enhanced Linux; see `http://www.nsa.gov/selinux`). This Linux implementation is actually a combination of type enforcement (TE), role-based access control (RBAC), and optionally multi-level security (MLS). The main goal of SELinux is to offer flexibility in access and labeling decisions. Binary compatibility is provided with existing applications but both changes to existing kernel data structures (for labeling) and API extensions are required to support security-aware applications. As a consequence, SELinux is useful as a test system to evaluate possible additions or enhancements to the Linux kernel, but adoption appears difficult, in its present form, by the Linux community.

A radically different approach is used in the LOMAC prototype [Fraser 2000]. In this case, the idea is to minimize the total compatibility cost and still offer some useful protection benefit. The architecture is based on the combination of a loadable kernel module and the interposition at the system call interface to modify the operating system's behavior. From this viewpoint, it is very similar to REMUS. However, the security model is completely different. REMUS allows or denies the execution of critical system calls by looking at the contents of the ACD. LOMAC defines the concepts of *subject* (active entities that execute programs), *object* (passive entities that contain data), and *level* (labels indicating a relative level of integrity). The protection is provided by restricting the behavior of subjects in such a way that it is not possible to move potentially corrupted data from lower-level to higher-level objects. At configuration time, it is possible to define a LOMAC policy. This means specifying the number of levels in use and the mapping between existing objects and levels. It is difficult to evaluate the effort required to define a meaningful configuration.

In Sekar et al. [1990] a high-level specification language called Auditing Specification Language has been introduced for specifying normal and abnormal behaviors of processes as logical assertions on the sequence of system calls

and system call argument values invoked by the process. Although very elegant, this approach is less flexible than ours. Our starting point has been a detailed analysis of the interaction between privileged tasks and each system call. Then we have determined the set of checks required to execute each threat level 1 system call in a safe way. This set depends strongly on the specific system call. For instance, for `open` the set is a list of directories and files, whereas for `init_module` is a signature of file contents. We have a very simple interception and control mechanism (a set of "hooks" in the system calls and a set of specific rules collected in the ACD, respectively) that can be easily fit to any particular system call. The focus of our research is not the infrastructure but the contents of the ACD.

One of the distinguishing features of REMUS is the introduction of a "signature" to validate the contents of the loadable kernel modules (LKM). To the best of our knowledge, no other proposed solution for the control of system call behavior seems to consider explicitly the problem of LKM reliability.

Attacks based on the subversion of buggy privileged tasks have been around at least since 1980s (the *Internet Worm* exploited a buffer overflow in the `fingerd` daemon [Spafford 1989]) and many solutions have been proposed in the past to solve the problem. A quick review of alternative approaches (which do not control system call execution) concludes this section. Marking both data and stack regions as nonexecutable catches most cut-and-paste exploits of buffer overflow. A nonexecutable stack is readily implemented [Solar Designer 1997] since it introduces just minor side effects in most UNIX variations (e.g., Linux places code for signal delivery onto the process's stack). Note that there is no performance penalty and existing programs require neither changes nor recompilation. The situation is not so simple for the data region. It is not possible to mark it as nonexecutable without introducing major compatibility problems. Even if this could be solved, there is still the problem of attacks which instead of introducing new code, corrupt code pointers. This technique allows the execution of dangerous instructions that are already part either of the program or of its libraries [Wojtczuk 1998].

Recently, two compiler techniques have been proposed for introducing in the executable code "lightweight" checks on the integrity of the functions' return address. StackGuard [Cowan et al. 1998] detects and defeats stack smashing attacks by protecting the return address on the stack from being altered. StackGuard places a "canary" word next to the return address when a function is called. If the canary word has been altered when the function returns, then a stack smashing attack has been attempted, and the program responds by emitting an intruder alert into syslog, and then halts. To be effective, the attacker must not be able to "spoof" the canary word by embedding the value for the canary word in the attack string. StackGuard includes a number of techniques to prevent canary spoofing. StackGuard is implemented as a small patch to the gcc code generator, specifically the `function-prolog()` and `function-epilog()` routines. `function_prolog()` has been enhanced to lay down canaries on the stack when functions start, and `function_epilog()` checks canary integrity when the function exits. Any attempt at corrupting the return address is thus detected before the function returns.

The StackShield [Vendicator 1999] protection system copies the return address in an unoverflowable location (the beginning of the data segment) on function `prolog` and checks if the two values are different before the function returns. If the two values are different the return address has been modified so StackShield terminates the program or tries to let the program run ignoring the attack (risking at maximum a program crash). StackShield works as an assembler file processor and is supported by gcc/g++ frontends to automate the compilation. No code change or other special operations are required.

StackGuard and StackShield offer many nice features: minimum performance penalty, no change in existing code, and no constraint imposed on new code. The major limitation is that they protect against buffer overflows in the stack. Unfortunately, heap overflows are less common but by no means less dangerous than stack overflows [Conover and the w00w00 Security Team 1999]. The solution we propose is effective regardless of buffer location. Moreover, it has shown recently that it is possible to exploit buffer overflow vulnerabilities in the stack even in programs compiled with StackGuard or StackShield [Bulba and Kil3R 2000].

Last, but not least, it is necessary to take the following into account.

—gcc is NOT the only C compiler available.

—Checks introduced by a compiler are not selective: each function of any process is affected. Even if benchmarks of single applications do not show significant performance degradation, the impact on performance is unclear if the entire system software (kernel, libraries, utilities, etc.) is compiled with StackGuard (or StackShield).

6. CONCLUDING REMARKS

We have described how simple enhancements of an existing kernel code can make threats harmless that aim to subvert the execution of privileged application. These enhancements are supported by a detailed system call analysis which identifies a subset of system calls and a subset of tasks (the noninteractive privileged tasks) that need to be monitored to offer protection against local or remote attacks aimed at gaining immediate and full control of the target system. The results of the analysis appear relevant to reduce the monitoring overhead, and the development and maintenance effort. An additional interesting aspect is that our analysis could give important hints for developing more secure privileged applications. Indeed, a developer should pay special attention in the design and testing of applications that invoke critical system calls.

Based on this analysis, we have built two versions of the REMUS prototype, one implemented as a kernel patch and another as a loadable kernel module for the Linux operating system. When an attack is discovered, REMUS reacts and does not allow continued access by the associated process. The REMUS prototype has been in use since Spring 2000 in two organizations and no fault has been reported by the users. In the short term, we expect to add “response” capabilities to our attack detection mechanism. The objective is to develop a kernel subsystem for real-time intrusion tolerance. Simple systems such as the cage [Bellovin and Cheswick 1994] have been already used in the past to analyze

the intruders' activities in progress without letting them notice it. However, those systems are not activated on the fly during the intrusion attempt. The real-time intrusion handling mechanism we have in mind requires the migration of the offending process to a distinct system designed to reproduce the original environment as faithfully as possible. Such an intrusion-handling subsystem seems useful for a couple of reasons. If a user process P comes under suspicion for its behavior, further investigation may show that what looked suspicious was actually unusual but legitimate activity. If this is the case, denying P access could result in unwarranted denial of service. It is even possible that an attacker is intentionally spoofing P to provoke a denial of service response against P. On the other hand, if P proves guilty, immediately denying access to the system may mean losing the opportunity to gather more information that would help identify the attacker and the objectives of the attack, and analyze possible collusions among distinct intruders.

In the medium term, we would like to port the REMUS system we developed for Linux to other UNIX flavors whose source code is available (xxBSD, Solaris).

APPENDIX: THE `check_rootproc` FUNCTION

```
int check_rootproc(struct inode *ino) {
   int cont=0,iproc=0,error=0;
   suidp_t * suidproc;
   efile_t f;
   suidp_t p;

    if ((IS_SETUID_TO_ROOT(current))||(IS_A_ROOT_DAEMON(current)))  {
      for (;cont<admitted.total;cont++) {
        if((admitted.lst[cont].efid.device==ino->i_dev)&&
           (admitted.lst[cont].efid.inode==ino->i_ino)) {
            if((admitted.lst[cont].efid.size==ino->i_size)&&
               (admitted.lst[cont].efid.modif==ino->i_mtime)) {
                suidproc=admitted.lst[cont].processes;
                for (iproc=1;iproc<=admitted.lst[cont].proc_nr;iproc++)  {
                     if (!strcmp(suidproc->suidp_id.comm,current->comm)) {
                        suidproc->suidp_id.count++;
                        return PSA;
                     }
                     if (iproc<admitted.lst[cont].proc_nr) {
                        suidproc=suidproc->next;
                    }
                }
            } else {
               error=EFNA;
               goto file_exe_unauthorized;
            }
        }
      }
      error=EXENA; /* EXE is not in the database */
      goto file_exe_unauthorized;
    }
    return PNS; /* the process is not setuid to root or root daemon */
```

```
   file_exe_unauthorized:
                    f.efid.device=ino->i_dev;
                    f.efid.inode=ino->i_ino;
                    f.efid.size=ino->i_size;
                    f.efid.modif=ino->i_mtime;
                    strncpy(p.suidp_id.comm,current->comm,
                            sizeof(p.suidp_id.comm));
                    p.suidp_id.count=1;
                    do {
                      while (writer_pid!=0){
                        cli(); /* interrupt disabled */
                        if (writer_pid!=0)
                          interruptible_sleep_on(&pid_queue);
                        sti();
                      }
                    } while (!atomic_access(&writer_pid,current->pid));
                    /* start of critical section */
                    do_setuid_put(&(f.efid),&(p.suidp_id),FAILURE);
                    writer_pid=0; /* end of critical section */
                    atomic_access(&writer_pid,0); /* release of the lock */
                    return error;
}
```

ACKNOWLEDGMENTS

The authors gratefully acknowledge the anonymous referees for their helpful comments. We would like to thank Trent Jaeger for illuminating discussions. A number of students have been involved in testing various parts of the REMUS prototype. Among them Ivano Alonzi has been responsive in a special way.

REFERENCES

ALEPH ONE 1996. Smashing the stack for fun and profit. *Phrack Mag. 7*, 49.

AMES, S. R., GASSER, M., SCHELL, R. R. 1983. Security kernel design and implementation: An introduction. *IEEE Computer 16*, 7, 14–22.

BADGER, L., STERNE, D. F., SHERMAR, D. L., AND WALKER, K. M. 1995. A domain and type enforcement UNIX prototype. In *Proceedings of the Fifth USENIX UNIX Security Symposium* (Salt Lake City, June).

BELLOVIN, S. M. AND CHESWICK, W. R. 1994. *Firewalls and Internet Security*. Addison-Wesley, Reading, Mass.

BERNASCHI, M., GABRIELLI, E., AND MANCINI, L. V. 2000. Operating system enhancements to prevent the misuse of system calls. In *Proceedings of the Seventh ACM Conference on Computer and Communications Security* (Athens, Greece, Nov.), 174–183.

BULBA AND KIL3R 2000. Bypassing StackGuard and Stackshield. *Phrack Mag. 10*, 56.

COMER, D. E. AND STEVENS, D. L. 1998. *Internetworking with TCP/IP*, vol. 3. Prentice-Hall, Englewood, Cliffs, N.J.

CONOVER, M., AND THE *w00w00* SECURITY TEAM 1999. w00w00 on heap overflows. http://www.w00w00.org.

COWAN, C., WAGLE, P., PU C., BEATTIC, S., AND WALPOLE J. 2000. Buffer overflows: Attacks and defenses for the vulnerability of the decade. http://www.cse.ogi.edu/DISC/projects/immunix.

COWAN, C., PU C., MAIER, D., HINTON, H., BAKKE, P., BEATTIE S., GRIER A., WAGLE, P., AND ZHANG, Q. 1998. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. *USENIX UNIX Security Symposium* (San Antonio, Jan.).

FINE, T. AND MINEAR, S. E. 1993. Assuring distributed trusted Mach. In *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy* (May).

FRASER, T. 2000. Low water-mark integrity protection for COTS environments. In *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, May).

GHORMLEY, D. P., PETROU, D., RODRIGUES, S. H., AND ANDERSON, T. E. 1998. SLIC: An extensibility system for commodity operating systems. In *Proceedings of the USENIX Annual Technical Conference* (New Orleans, June).

GOLDBERG, I., WAQNER D., THOMAS, R., AND BREWER E. A. 1996. A secure environment for untrusted helper applications. In *Proceedings of the Sixth USENIX UNIX Security Symposium* (San Jose, Calif., July).

INTRUSION DETECTION SYSTEMS 1999. http://cerias.purdue.edu/coast/intrusion-detection.

LUNT, T. 1993. A survey of intrusion detection techniques. *Comput. Sec. 12*, 4 (June), 405–418.

MINEAR, S. E. 1995. Providing policy control over object operations in a mach based system. In *Proceedings of the Fifth USENIX UNIX Security Symposium* (Salt Lake City, June).

MUDGE 1996. How to write buffer overflows. http://www.l0pht.com/advisories/bufero.html.

RIVEST, R. 1992. The MD5 message-digest algorithm. RFC 1321.

SEKAR, R., BOWEN, T., AND SEGAL, M. 1999. On preventing intrusions by process behavior monitoring. In *Proceedings of the First Usenix Workshop on Intrusion Detection and Network Monitoring* (*ID*) (Santa Clara, Calif., April).

SOLAR DESIGNER 1997. Non-executable user stack. http://www.openwall.com/linux.

SPAFFORD, E. H. 1989. Crisis and aftermath. *Commun. ACM 32*, 6 (June), 678–687.

STEVENS, W. R. 1998. *Unix Network Programming*, Second ed., Prentice-Hall, Englewood, Cliffs, N.J.

VENDICATOR 1999. Stack Shield: A stack smashing tecnique protection tool for Linux. http://www.angelfire.com/sk/stackshield.

WOJTCZUK, R. 1998. Defeating Solar Designer non-executable stack patch. Bugtraq mailing list: January 30. http://www.securityfocus.com/bugtraq.

Received February 2001; revised December 2001; accepted December 2001